Bolt Beranek and Newman Inc.

(12)

LEVEL II

Report No. 4660

Development of a Voice Funnel System

Quarterly Technical Report No. 8
1 May 1980 to 31 July 1980

May 1981

Prepared for:
Defense Advanced Research Projects Agency

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

REPORT DOCUMENTATION PAGE		READ INSTRUCTIONS BEFORE COMPLETING FORM
1. REPORT NUMBER	2. GOVT ACCESSION NO. AD-A099 873	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) (6) DEVELOPMENT OF A VOICE FUNNEL SYSTEM. (9) QUARTERLY TECHNICAL REPORT No. 8, 1 May-31 Jul 80,		5. TYPE OF REPORT & PERIOD COVERED Quarterly Technical
		6. PERFORMING ORG. REPORT NUMBER (14) BBN-4660 ✓
7. AUTHOR(s) (10) R. D. Rettberg		8. CONTRACT OR GRANT NUMBER(s) (15) MDA903-78-C-0356, ✓
9. PERFORMING ORGANIZATION NAME AND ADDRESS Bolt Beranek and Newman Inc. 50 Moulton Street Cambridge, MA 02238		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS ✓ARPA Order No.-3653
11. CONTROLLING OFFICE NAME AND ADDRESS Defense Advanced Research Projects Agency 1400 Wilson Boulevard Arlington, VA 22209		12. REPORT DATE (11) May 1981
		13. NUMBER OF PAGES 32 (12) 36
14. MONITORING AGENCY NAME & ADDRESS (if different from Controlling Office)		15. SECURITY CLASS. (of this report) UNCLASSIFIED
		15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

DISTRIBUTION UNLIMITED

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

18. SUPPLEMENTARY NOTES

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Voice Funnel, Digitized Speech, Packet Switching, Butterfly Switch, Multiprocessor

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This Quarterly Technical Report covers work performed during the period noted on the development of a high-speed interface, called a Voice Funnel, between digitized speech streams and a packet-switching communications network.

DD FORM 1 JAN 73 1473 EDITION OF 1 NOV 65 IS OBSOLETE

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE (When Data Entered)

a00100 609

DEVELOPMENT OF A VOICE FUNNEL SYSTEM

QUARTERLY TECHNICAL REPORT NO. 8
1 May 1980 to 31 July 1980

May 1981

This research was sponsored by the
Defense Advanced Research Projects
Agency under ARPA Order No.: 3653
Contract No.: MDA903-78-C-0356
Monitored by DARPA/IPTO
Effective date of contract: 1 September 1978
Contract expiration date: 30 November 1980
Principal investigator: R. D. Rettberg

Prepared for:

Dr. Robert E. Kahn, Director
Defense Advanced Research Projects Agency
Information Processing Techniques Office
1400 Wilson Boulevard
Arlington, VA 22209

Table of Contents

1. Introduction

This Quarterly Technical Report, Number 8, describes aspects of our work performed under Contract No. MDA903-78-C-0356 during the period from 1 May 1980 to 31 July 1980. This is the eighth in a series of Quarterly Technical Reports on the design of a packet speech concentrator, the Voice Funnel.

2. Microcode Development System

At BBN we have developed microcode for a number of microprogrammable machines, and our experience has been that writing microcode, especially horizontal microcode, is a tricky business. Conventional microassemblers are primitive and encourage an obscure programming style. Moreover, they provide little help in detecting microcode errors. Recent efforts in this area have attempted to introduce stylistic principles borrowed from higher level languages. Although interesting, most of these systems are designed around a particular micromachine and trade object code efficiency for programming ease -- something we cannot afford to do. Instead, we have chosen to implement a microassembler whose language 1) encourages a clear programming style (and encourages commenting of code), 2) includes a concise bus transfer notation, and 3) has checks for several common classes of coding error.

The Butterfly Processor Node hardware includes five different, independently programmed micromachines; each has its own microinstruction set. Although the machines are quite different, the microassembler we have developed is general enough to be used for all of them. The assembler is designed to be extensible in a number of dimensions:

- Machine-dependent symbols are defined externally by a header file.
- The word size is an explicit parameter to the micro-

assembler.

- Most validity checking is based on explicit type information.
- The assembler is written using the UNIX programming tools LEX and YACC, permitting its syntax to be easily modified or extended.
- The internal structure of the microassembler has been designed modularly so that new facilities and new microinstructions can be readily installed.

Only two versions of the assembler are necessary to handle all five instruction sets, and these versions share a large amount of identical code. Thus the assembler is quite general and should be useful for other projects.

Many microcode assemblers are based on the concept of key words, which specify that a particular operation should be performed by the hardware during the current microcycle. Sometimes additional arguments, such as counts or addresses, may be supplied with the keyword. One major difference between normal assembly languages and microassembly languages is that several such keywords are typically specified in each microinstruction. The Butterfly microassembler provides for simple keywords directly, and makes it easy to add functions (keywords which take arguments). Functions are written in C and included in a "user augmentation package" which is linked into the assembler itself.

Micromachines commonly consist of a number of registers, interconnected by a number of buses. In such machines, the idea of a bus transfer operation is quite natural. A unique feature of the Butterfly microassembly language is the way in which bus transfers are denoted, and the associated validity checking. For example, a common operation might be to route the results of an arithmetic computation to a data bus, and from there to a memory bus, and finally to store the result in a register in the memory system. In the PNC micromachine this operation would be written as follows:

```
alu -> D -> mAD -> ADr
```

Each arrow replaces a keyword in a more conventional language. A great advantage of this approach, besides its obvious clarity, is the checking which can be provided; the alu is declared to be a valid data source, and the ADr a valid data sink. The intervening buses are declared to be valid sources only if they have also been used as sinks. Then the assembler can verify that each transfer has a valid source, and that no sink is used more than once per microinstruction.

The final interesting feature we have implemented involves a unique form of error checking provided by the microassembler. In complex micromachines there frequently are conflicting functions available to the microprogrammer. The use of one of these functions prevents or restricts the use of one or more other functions. Here are some examples:

- A single field in the microinstruction word is decoded to select one of several functions. Obviously, only one of these functions can be specified in a given microinstruction.

- Two otherwise independent functions require arguments which overlap in the microinstruction word. Only one of these functions can be specified, unless the arguments happen to match in their overlapping sections.

- Although two functions can be specified independently in the microinstruction word, the hardware cannot logically perform both at once. For example, the memory system cannot accept an address and an address strobe in the same microinstruction.

- Timing considerations are such that, although the hardware has the capability to attempt a specified complex operation, it cannot be completed successfully within a single microcycle.

- Some operations take more than one microcycle to complete, so the legality of a microinstruction is, in general, affected by previous microinstructions.

Although we do not know of any technique to check for all of the problems listed above, we have implemented features which do catch many of them.

All of the numbers, keywords, and expressions used in the Butterfly microassembler are assigned data types. Types are composed of the values of a number of more or less independent "attribute fields," which may or may not be present in each particular case. The most important attributes are numeric, which allows arithmetic operations; unital <*> , which prohibits

<*> The word unital is related to the concept of a unit, something complete in itself. Unital symbols cannot be combined to form other symbols or expressions.

arithmetic or logical operations; field-value, which differentiates keywords from other symbols, and explicit types for the arguments of certain functions. For example, statement labels have the following attributes: LABEL, numeric, not-unital, not-field-value. The branch and call functions check their argument for the appropriate attribute (in this case, LABEL), and would reject, for example, branching to the value of a keyword. On the other hand, if a label were to be used in place of a keyword, it would be rejected based on its not-field-value attribute.

A microinstruction is written as a number of separate clauses, and it is valuable to check that these clauses do not conflict with each other. This is done by using extra bits to specify which fields are being set by each clause. Each numeric value in the microassembler is kept in two parts, a mask section which specifies which bits are significant, and a normal value section. Another way of looking at this is that numeric values are kept in tri-state, that is, each bit is a zero, a one, or a don't-care. When the assembler combines clauses to form a microword, it checks to be sure that no conflicts exist. A conflict would exist only if a bit were set to both one and zero by different clauses within the microword. Overlaps are allowed if the values for the bits in question agree.

Because of the unique features of this microassembler, we have chosen to include the following fairly detailed description, excerpted from the microprogrammer's manual prepared for the Process Node Controller (PNC) micromachine. Some sections are complete, to give the reader the full flavor of what we have done; others have been shortened to remove unnecessary detail, and a few less interesting sections have been omitted entirely.

2.1 Overview of the Processor Node Controller

The Processor Node Controller (PNC) is a micro-machine which provides the interface between the other components of the Processor Node and implements several operations performed under program control from the main processor, an MC68000. Major services provided by the PNC include:

1. all MC68000 memory accesses, including main memory, remote memory, EPROM, I/O device registers, memory management registers, and various other internal registers which appear in the MC68000's address space;
2. main memory refresh;
3. all I/O system memory accesses, interrupts, posts, dequeues, etc.;
4. all Butterfly Switch functions including three independent functions: transmit, receive, and reply; transfers of bytes, words, and blocks of data; and transmission interrupts and other special messages across the Butterfly Switch;
5. special MC68000 support functions implemented by recognition of special addresses in main memory. Examples of these are a 32-bit real time clock, the dual queue mechanism, etc.

The PNC is organized as a 64-bit micromachine. The microcode words are divided into fields in such a way that several functions can be controlled at one time. An AMD 2911 microsequencer selects the microinstruction to be executed next. Services are provided by the PNC in response to microinterrupt requests. When no requests are pending, the PNC executes an idle loop. Requests are detected in hardware and cause microinterrupts to the start of the appropriate service routine, which returns when the request has been satisfied. Microinterrupts are normally inhibited during service routines.

The microassembler for the PNC runs on a VAX 11/780 computer running the UNIX operating system and is described in detail below. The following example gives an idea of what the language is like:

```
;This is a sample microinstruction.
;Semicolons start comments which extend to end-of-line.

idleloop:               ;a label starts the microword
        at(0x113),      ;this puts 'idleloop' at location 113 hex
        epi, emi,       ;epi/emi together enable microinterrupts
        br(idleloop),   ;'idleloop' branches back to itself
        .               ;and other things happen as well
        .
        .
```

All of this is a single microinstruction which occupies one word in micromemory. We will come back to this example later.

2.2 Basic Rules and Syntax for the PNC

2.2.1 Syntax and Arithmetic

This section describes how the microcode assembler syntax works and conventions for using it, and specifies some standard names for buses, latches, signals, ALU functions, etc. Symbols start with a letter and consist of at most ten letters, numbers, and any of the following characters: period (.), underline (_), and number sign (#). In this memo we use the symbol number sign (#) to indicate a numeric quantity.

Since the microassembler is used for various micromachines, it must be customized by a header file describing a specific micromachine. The initial symbol table for the PNC is specified in the file "pnc.h", and must be included in all programs written for the PNC. Additional symbols can be defined using either an equals sign (=) or a colon (:); see examples below.

Basically, the microassembler assembles microinstructions which have values and are placed at specific locations in micromemory. Microinstructions are made up by combining a sequence of clauses as described below. Each clause comprises either a single term or a number of terms combined by operators. The terms themselves are integers, predefined symbols, functions, or parenthesized expressions. Integers are decimal by default; prefixing 0x specifies hexadecimal, 0b specifies binary, and a simple leading 0 specifies octal. A number of operators are available:

Binary operators:

```
+ (plus), - (minus), * (times), / (divided by),
% (remainder), >> (right shift), << (left shift),
& (and), | (ior), ^ (xor),
-> (bus transfer; see below),
<* (special left shift; see below)
```

Unary operators:

```
- (two's complement),
~ (one's complement),
! (logical negate; see below)
```

Precedence is the same as in the programming language C, with <* at the same level as <<, and -> below |. Symbols can be defined to restrict the ways in which they can be combined: unital symbols cannot be combined with other symbols; non-numeric symbols cannot be combined with numbers; some symbols have a specific type, such as LABEL, and cannot be combined with symbols of other types.

2.2.2 Combining and Checking Clauses

The PNC microword is 64 bits wide and is divided into a number of (sometimes overlapping) fields. The microcode assembler assembles each clause (which may define several fields) independently, then IORs the clauses to form the microword. Two kinds of checking are performed automatically to help detect bugs. The first check is for inconsistent overlapping fields, the second for buses which have a sink specified, but have no source.

All clauses are checked against each other for conflicts as they are being combined. To allow for this checking, the terms which make up each clause specify a mask as well as a value. The mask specifies which bits are being defined (the fields) and the value is limited to just those bits. Terms are specified in one of three ways:

1. symbols specify a mask and a value. For example:

   ```
   ret = mvp(0x78000, 0x50000)   ;return from subroutine
   ```

   (The function "mvp" generates a mask-value pair from the values of its arguments.)

2. integers, which have an intrinsic value; -1 is used as their mask.

3. functions, which take arguments and which are mostly PNC-dependent. Functions are written in C, are called from the assembler with their arguments evaluated as clauses, and generate a term or an error message. For example:

   ```
       br(address,cond2,cond1)    ;branch conditionally
   ```

   or specifically,

   ```
       br(0x34,zero,neg)
   ```

   This produces a multi-way branch depending on the ALU at the start of the micro-instruction. The definition of the "br" function is specified more fully in section 2.3.

Given a new field, the assembler first checks that each one bit in its value has the corresponding one bit set in its mask; it then checks the field for conflicts against each previous field, as follows:

> XOR the new value with the old value, then AND with both the new and old masks; a non-zero result indicates a conflict.

Fields may overlap as long as the overlapping bits are the same in both fields. Bus transfer functions also maintain and check words which contain bits for each bus and latch, specifying whether or not they are currently a source or sink. These are reset at the start of each microword.

The special operator "!" (exclamation point) specifies that a term should be logically negated (set to 0). This is used to assert that some function cannot be used or is not being used. For example:

```
!eras  ( = 0x800000000, 0 )     ;must not set eras
```

"!" returns the mask unchanged, with a value of zero; if the value was already zero you will get an error message.

Terms are combined into fields by various operators. These must combine both values and masks to create a result which reflects the user's intent. Here is a summary:

```
-a:     mask = mask(a), value = -a & mask(a)
~a:     mask = mask(a), value = (a ^ -1) & mask(a)
!a:     mask = mask(a), value = 0

a|b:    mask = mask(a) | mask(b), value = a|b
a^b:    mask = mask(a) | mask(b), value = a^b
a&b:    mask = mask(a) | mask(b), value = a&b
a>>b: mask = mask(a), value = a>>b
a<<b: mask = mask(a), value = a<<b
a<*b: mask = mask(a) << b, value = a<<b
a+b:    mask = mask(a) | mask(b), value = a+b
a-b:    mask = mask(a) | mask(b), value = a-b
a*b:    mask = -1, value = a*b
a/b:    mask = -1, value = a/b
a%b:    mask = -1, value = a%b              ;remainder
```

The bus transfer operator is special and can generate several separate fields. For example:

```
D -> mAD -> Ar          ;each arrow generates a field.
```

The assembler evaluates the terms before and after the arrow and calls the bus-transfer function which looks up the specified transfer in the symbol table. In this case, the two fields generated would be:

```
D -> mAD     ( = 0x0, 0x0 )          ;hardware default
mAD -> Ar    ( = 0xf00000000000, 0x700000000000 )
```

In summary, microinstructions are specified by a sequence of fields, with comments as appropriate. The fields are normally specified in the order in which they will happen during the microinstruction. Bus transfers send a source to a sink. Some things are always sources, such as eprom; others, such as D (the data bus), are initially sinks, but once their source is specified they become sources themselves. It is illegal to use a sink as a source, or a source as a sink. Therefore the order in which bus transfers are specified is often critical. The order in which the other fields are specified is up to the programmer, but a few conventions have been adopted. For example, the branch field goes first to emphasize that conditional branches test at the start of the microcycle; comments include any unusual assumptions or timing calculations.

When defining symbols for the various PNC functions, we have chosen to mix upper and lower case letters to make the abbreviations easier to read. In many cases, the names match the corresponding signal names used on the prints. "r" usually means register, "A" means address, "D" means data, etc.

When all fields have been combined, a post-processing function fills in unspecified fields, complements inverted fields, and could do any further checking which might prove desirable.

2.2.3 A Sample Microinstruction

Now let us get back to the example we started above. This is the full-blown version of the idle loop, as currently implemented:

```
; This microinstruction is executed whenever the PNC is not
; executing microinterrupt service routines.  It may only be
; executed when the aux is set to its default (0x07), which gates
; mmRl and mmRh to aD.  It anticipates an MC68000 read/write
; request, and prepares to service any of a variety of such
; requests (main memory, EPROM, etc.).

idleloop:
     at(0x113),              ;anywhere
     epi, emi,               ;enable MC68000 and misc interrupts
     br(idleloop),           ;loop until interrupted
     !mmu9,                  ;select the mem man relocation word
     assert(mmR -> aD),      ;the aux must have been set correctly
     aDl -> mA,              ;give bits 19.16 of address to memory
     aDh -> pmA -> Dh,       ;compute bits 15.8 of address and
     cpuAl -> Dl,            ; combine in bits 7.0 (bit 0 is unknown),
     D -> mAD -> ADr,        ; give low 16 bits of address to memory
     cpuOP,                  ;tell memory to latch the CPU r/w request
     mAD -> Ar,              ;move the phys addr into the A register
     move(D, r0)             ; and into r0
```

The intent of this microinstruction is to be ready to process a memory request from the MC68000 as fast as possible. The microinstruction anticipates an MC68000 memory access, and performs the initial portion of the set up, which does not adversely affect the processing of other microinterrupts. Other microinterrupts do not need most of the idle loop functions.

2.3 Branching, Subroutines, and Interrupts

We discuss below how branches, subroutine calls, and interrupts work, and how to decide where to place each microinstruction in memory. We have chosen not to automate this last function in the assembler, but have designed some simple features and conventions to help the microprogrammer with this task. A preprocessor, 'amc', is available to help assign microcode addresses. It reads a complete microprogram, checks it for certain errors, and edits the 'at' commands so as to specify valid, non-overlapping instruction placement and to compact the program as much as possible. It also produces a symbolic memory map, showing microinstruction locations and dependencies. See section 2.6 for further information.

In addition to interrupting such things as branch and call clauses, amc uses other more unusual clauses, which we call assertions. These are functions which do not in themselves cause things to happen, but rather express the programmer's intent that they will happen or must happen. Sometimes assertions are used by the microassembler in the process of checking for errors of various kinds.

After discussing branching, we specify the details of the bus transfer operations, then discuss the alu, and finally get into some of the other PNC control bits and fields in the microword.

While each microinstruction is being executed, the next microinstruction's location is being determined, and the instruction itself is being fetched. This function is performed by a microprogram sequencer and controlled by the current microinstruction. The programmer may specify branching directly to a 10-bit address, use the current address plus one, use the address from the top of the sequencer's stack, or use an internal address register (the uAr). The PNC organization is based on the idea that almost all of its functions will be performed by microinterrupt routines.

If the microinterrupt system is enabled and an interrupt request is pending, the address specified by the programmer will be placed in the microprogram counter, but the address specified by the microinterrupt system will be used to select the next microinstruction. In effect, this causes that instruction to be executed as if it had been wedged in between two ordinary instructions. It can, and normally will, use the microsequencer subroutine call facility to call the rest of a multi-instruction interrupt routine; returning from a microinterrupt is the same as returning from an ordinary subroutine. MC68000 microinterrupt service routines sometimes enable I/O system interrupts. The microsequencer stack has a total of four locations; one is reserved for MC68000 microinterrupts, one for other (including I/O system) interrupts, and two are available for calling ordinary subroutines (three stack locations are available).

The location at which a microinstruction is loaded may be constrained in a number of ways:

1. It is the first instruction of a microinterrupt routine. The address of each microinterrupt routine is determined in hardware by a PLA. Since these instructions ordinarily specify a branch address (see below), changes to the interrupt routines themselves do not matter and the addresses are chosen to keep the PLA definition compact.

2. It must follow a particular microinstruction. This occurs when the previous microinstruction uses the microprogram counter (pc) plus one addressing mode, either directly or by calling a subroutine which returns normally.

3. It is the destination of a conditional branch. There are two independent conditions which may be specified, one which selects bit 0 (even/odd addresses), the other, bit 1.

4. A hardware reset of the PNC causes the microinstruction at location zero to be executed.

5. Since the litA field overlaps the branch address field, an instruction which must branch and load a literal can be written only if its destination can match the value of the literal. Where this technique is used, the location of the destination instruction must be fixed.

A number of functions are provided which help the programmer to specify and document where microinstructions are to be loaded, and to control branching and stack operations.

- "at(address)" specifies the current location of a microinstruction. If omitted, the microinstruction is placed at the previous address plus one.

- "fixed_at(address)" specifies that a microinstruction must be placed at a specific, fixed address.

- "interrupt" asserts that this microinstruction is the first instruction in a microinterrupt routine. The instruction must also include the fixed_at function. Specifying "interrupt" does not affect the microassembler directly, but is required for amc to work correctly.

- "case(cond2,cond1)" asserts that the microinstruction is branched to under the listed condition(s). For example,

```
case(!zero,neg)
```

  asserts that this instruction is the destination of a conditional branch, and that the alu will have been less than zero when control comes here through that path. This assertion does nothing in the microassembler except to generate an error if the actual address does not meet the stated requirements of the target for the stated condition (i.e., is not equal to 2 mod 4, see below). Either condition may be left blank.

- "nocase(cond2,cond1)" is always used with a case statement. It asserts that the specified case cannot occur. For example,

  ```
  case(!zero,neg),        nocase(zero,neg)
  ```

  the nocase clause asserts that the zero&neg case is impossible, and that no microinstruction has been provided to handle it. This assertion does nothing in the microassembler unless its arguments are invalid. It is used by amc to suppress spurious warning messages.

- "br(address,cond2,cond1)" branches to the specified address or successive addresses, depending on the specified conditions. If both conditions are omitted, the branch is unconditional. If only cond1 is specified, "address" must be even and the branch will go to "address"+1 if the condition is absent or false. If only cond2 is specified, the low order two bits of "address" must be either 00 or 01 and the branch will go to "address"+2 if the condition is absent or false. Finally, if both conditions are specified, "address" must be divisible by 4 and the branch will go to "address" if both conditions are present, +1 or +2 if only one is present, and to "address"+3 if both conditions are false or absent. See below for a list of conditions which may be tested.

- "br(uAr)" and "br(stack)" are special forms of br (and also call) which use the contents of the uAr, or the top word on the stack, to specify the next microinstruction to execute.

- "next" asserts that this microinstruction "branches" to the next instruction, i.e., uses pc+1 mode. This mode is the default and sometimes is required because a branch address would use conflicting fields. "next" can be used to emphasize that branching is impossible.

- "call(address,cond2,cond1)" works exactly like br(,,) except that pc+1 is stored on the top of the 2911 stack.

- "ret" branches to the location on the top of the stack and pops the stack. Because of a field overlap, if 'literal -> D bus' is used, bit 15 of the literal must be a zero.
- "setuAr(address,cond2,cond1)" specifies an address just as the similar form of br does, but loads the 10-bit result into the uAr. This function uses the same bits in the microword as br and call do, so pc+1 addressing mode is normally used to avoid conflicts.

There are a number of conventions which can help to make the microcode easier to read, understand, and modify. The use of assertions is one example. With respect to program flow, the code can be organized in linear fashion by function, rather than in the order in which it is loaded into the PNC. This requires many instructions to contain the "at()" function, and requires the programmer to consult a chart of available locations in micromemory to decide on locations to use, or to use amc to assign locations instead.

Conditional branches involve one (or two) out of the following list of conditions. The conditions are named in such a way that the branch will 'skip' if false; an 'n', meaning 'not', reverses the stated sense if there otherwise would be no good name for the condition.

Cond1 conditions - if condition is false, branch to address +1.

```
        Ar0          ;test address register (Ar) bit 0
        RxHCE        ;receiver header checksum error
        Ar2
        pwrok        ;power voltage is greater than 20 volts
        RxRME        ;receiver request message error
        TxSRM        ;transmitter request message buffer full
        npncBmas     ;not [processor node is IO bus master]
        nsDack       ;no [IO bus (synchronized) data acknowledge]
        nsAs         ;no [(synchronized) address strobe (bus grant)]
        wordOP       ;MC68000 word operation requested
        TxFIU        ;transmitter Fifo in use
        RxAME        ;receiver acknowledge message error
        neg          ;alu is negative
        carry        ;alu carry bit
```

Cond2 conditions - if condition is true or 1, branch to address +2.

```
        nPerr        ;no [parity error from last memory access]
        Ar1          ;address register (Ar) bit 1
        Ar3
        zero         ;alu is zero
        nsHALT       ;not [synchronized state of MC68000 halt line]
        ACCviol      ;access violation (from mem man hardware)
```

The standard microcode includes the specifications for and addresses of all the microinterrupts which can occur. In this document we will discuss the microinterrupt system in more general terms. There are two independent sets of microinterrupts, the processor interrupts and the miscellaneous interrupts. All processor interrupts go to addresses of the form 0x1yy. Examples of such interrupts are memory read/write requests (including main memory, EPROM, etc., as separate interrupts), dual queue requests, real time clock requests, etc.

The miscellaneous interrupts involve mainly the switch and the I/O bus interface. These interrupts can be divided into two classes: the three switch functions (send message, receive message, receive reply), and all the rest. The switch functions are organized as finite-state machines, with up to 16 states each. When a miscellaneous interrupt condition arises, the PNC goes to a location of the form 0x0yz, where y depends only on the condition, but z is selected from one of the four 4-bit entries in the return address register file (rA#). One of these entries is used for each of the three finite-state machines, and is explicitly set to the new state during state transitions; thus each state is an independent microinterrupt routine. The fourth entry is held at zero and is selected by all other miscellaneous interrupts; i.e., these have interrupt locations of the form 0x0y0.

2.4 Bus Transfer Operations

The bus transfer operator (->) is used to indicate that a source of bits is to be gated to a bus, latch, register file, or other data sink. Some things can be sources, some sinks, some either, some both, and some vary depending on how they have been set up. When literals are used as sources, they may come from any of several fields within the microword, depending on their sink. Some bus transfers are latched and remain in effect until modified; the "assert(x -> y)" function states that the current microinstruction depends on one of these latched transfers. The latch involved (the aux register) has a default setting to which it must be restored. Some sources supply only a subset of the bits required by a sink. For example, the aD bus is composed of two halves, aDh and aDl, which are separate sinks. If aD is used as a source, both aDh and aDl must have been used as sinks; however, aDh or aDl may be used as sources individually if desired, even though their sink would normally require that both be set up.

Because of this complexity, we have made the bus transfer function table-driven, with a minimum of special-case checking. The organization of these tables is beyond the scope of this report. We require that each bus transfer be specified individually, that is, not to allow one bus transfer function, say "a -> c", to imply two real transfers, "a -> b, b ->c". However, these can be abbreviated "a -> b -> c". There would be no reason why "a -> c" could not be defined directly, if that would prove desirable.

The alu can be a source of data, can operate independently, or can use data on the D bus. Bus transfers which set up the D bus for the alu must precede the alu specification, while transfers which use a value from the alu must follow it. The timing requirements for bus transfers, particularly ones involving the alu, can be critical; we have considered adding timing information to the bus transfer data and checking for violations, but the problem has not been serious enough to warrant the effort that would be required.

2.5 ALU Operation in the PNC

The ALU (4 2901s) is complex enough that special functions have been incorporated into the user augmentation package to generate its control fields in the microinstruction. The ALU includes sixteen registers (r0-r15), and a special register Q. The microinstruction must specify an ALU function (for example, subtract), what operands to use, and what to do with the results.

Here are some examples:

```
add(r1,r2, r2),             ;(r1 + r2) -> r2, and
alu -> D,                   ; also to the D bus.
```

The second line, which may be omitted, will gate the alu output to the D bus.

```
r8 -> D,                    ;old value of r8 goes to the D bus
add(r8,r8, r8),             ; and double r8.
```

This gates the old value of an alu register to the D bus. There are many important restrictions imposed by the 2901s; these are discussed below.

The general format of a function is:

```
function(arg1,arg2, destination)
```

The following functions are available:

```
add (arg1 + arg2), sub (arg1 - arg2)     ;arithmetic functions
ior (arg1 ¦ arg2), and (arg1 & arg2),    ;logical functions
xor (arg1 ^ arg2), nxor (~arg1 ^ arg2),
mask (~arg1 & arg2)               ;remove bits in arg1 from arg2
```

All arithmetic is two's complement. Some special-purpose functions are defined in terms of the above:

```
move(0, dest)          ; and(0,dest, dest)
move(arg, dest)        ; ior(arg,0, dest)
invert(arg, dest)      ;nxor(arg,0, dest)
comp(arg, dest)        ; sub(0,arg, dest)
```

In general, the arguments may be one of the following:

```
r0-r15                 ;alu registers,
D,Dh,Dl                ;the D bus,
Q                      ;the alu Q register, or
0                      ;the constant 0 (the default)
```

For arithmetic functions, arg2 may include a +1 clause (e.g., r6+1), which adds one to the value of arg2 before doing the function itself. ("0+1" may be abbreviated to "1".)

The ALU calculates the specified function, which may be sent to the D bus as mentioned above ("alu -> D"). In addition, the result may be stored back into the 2901s in a number of modes as specified in the destination field. The destination field may be one of the following:

```
r#              ;store result in r0-r15
Q               ;store result in Q
r#|Q, RL        ;rotate result and Q left, store in r# and Q
r#|Q, RR        ;rotate result and Q right, store in r# and Q
r#, RL          ;rotate result and Q left, store in r# only
r#, RR          ;rotate result and Q right, store in r# only
blank           ;ignore result (the default)
```

Only two different register numbers 'A' and 'B' can be specified in the microinstruction. Other restrictions, imposed by the hardware, are described in the Microprogrammer's Manual.

2.6 The 'amc' Preprocessor

One of the least interesting parts of writing microcode for the PNC and BIO 2901 micromachines is the task of assigning each microinstruction a location in memory which satisfies all the various constraints, is compact, and has no overlapping microinstructions. The preprocessor called 'amc' automates this task.

The 2901 micromachines have various constraints on instruction placement as discussed in section 2.3.

- Some microinstructions are explicitly fixed_at certain locations, for a variety of reasons. amc must leave these assignments alone, and work around them.
- Some microinstructions specify symbolic locations, such as 'at(abc+2)'. 'abc' itself may be fixed_at, movable, or be specified symbolically. [Restrictions: 'abc' must be defined before it is referenced, and must be defined as an instruction label directly, not by using '='.]
- Some microinstructions implicitly are associated with other microinstructions. For example, microinstructions which fall through to the numerically next microinstruction cannot be moved independently of that microinstruction. Another example is that matching cases of a conditional branch cannot be moved independently.

In general, related instructions form a dependency tree. The tree is fixed at a certain set of addresses if any of its elements has a numerical fixed_at clause. Otherwise the tree can be moved to any available set of locations which satisfy its other constraints; for example, conditional branch targets must be properly aligned.

amc is written as a preprocessor, which takes otherwise legal microassembler input files and translates their 'at' clauses to satisfy the constraints just discussed. It optionally produces a memory map which shows which microinstruction has been assigned to which location in memory.

2.6.1 How amc Works

amc is organized into three sections. The first (and longest) reads the input file(s), and parses them. It creates a data base entry for each instruction, which contains the following information:

- various bits which indicate the presence of 'next', 'br', clauses, etc.
- a chain word used to link together the elements of a dependency tree.
- the location within the file of the instruction's 'at' clause, if any.
- a pointer to the head of the dependency tree.
- the numerical location of the instruction, or its offset from the head of its dependency chain.
- the instruction's label as a character string.

'at' clauses are parsed, and explicit references to symbolic locations are noted. [Restriction: only the following syntax is allowed in at or fixed_at clauses: 'at()', 'at(number[+/-

number]...)', 'at(symbol[+/-number]...)'. Here number is a decimal, hexadecimal, or octal number in C format.] This processing sets up the explicit dependency trees; these are processed, and combined if necessary due to implicit dependencies, in the second section of amc.

The second section of amc locates the implicit dependencies and assigns actual addresses. It has four phases:

1. Instructions which contain no at or fixed_at clause are implicitly linked to follow the previous instruction in the input file. In addition, the dependency trees are flattened, that is, all the elements except the head of the tree are set to point to the head element. The other elements are chained to the head element through a linked-list. The head element always has a numeric (or null) at or fixed_at clause.

2. Phase two is the most complex phase, and in a sense is the heart of the program. For each instruction (i) which has an associated instruction, it locates that instruction and, if necessary, merges the two dependency chains. The two types of dependency checked for are the 'falls into' and the 'case' dependencies. If an instruction 'falls into' some other instruction, that instruction is located by the following procedure:

   - First, we search the existing dependency tree for an instruction at the correct offset from the head of the tree. Exit if found.

   - Otherwise we try to locate the related instruction by guesswork. This is based on two ideas. One, if values were specified for the instructions previously, a nearby instruction at the correct location is the one we want. Two, if no value was specified, we should try to insert such an instruction between existing instructions as well as possible. Thus, if instruction i does not specify a value, we guess a value for it, which would make the next instruction the one we want.

  - We start searching the file just after the head of the tree, looking for an ordinary instruction at the correct address, or an instruction with no address, immediately after instruction i. If we find something, we merge the two trees. The surviving tree is normally the one whose head came first in the file, unless the other tree was fixed_at, or had the only numerical address.

  - If we could not find anything, we print a warning message to the effect that the given instruction i implies a missing instruction. Sometimes this indicates a serious problem in the microprogram, but more often is simply the result of omitting impossible-to-reach cases, such as 'zero and negative'.

  As you might have guessed, a 'case' clause uses the same procedure, but tries to locate instructions at the appropriate offsets, which can be +1, +2, -1, or -2 from instruction i. Once an instruction has been located, various warning messages may result. These should be self-explanatory. The 'nocase' statement can be used to suppress warning messages caused by three-way branches, which otherwise would appear to imply a fourth case.

- When all the dependency trees have been built, the program sets up an array with an entry for each cell in micromemory. It then assigns all elements in fixed_at chains to the cell in which they must reside.

- Finally, amc tries to pack the rest of the trees into the remaining space. In theory this is an NP-complete problem, but amc takes a simple, basically linear-time approach to solving it. This could in theory, but does not in practice, fail to give acceptable results. It works as follows. amc locates the first open spot in memory and the first unassigned tree head. It attempts to put the tree at that point in memory; if all of the instructions in the tree will fit and have the proper alignment, it is happy. Otherwise, it tries again with the tree head going into the next available spot. It proceeds until the tree fits, or it runs out of memory. Then it starts over with the next tree. (This method is linear-time under the assumption that the average number of tries is bounded. This is true because of the high proportion of length-one, unconstrained trees in the actual data, and the fact that amc keeps track of the first open spot in memory and does not keep rescanning the totally allocated section of memory.)

The third section of amc rereads all the input files, and rewrites them. It changes only the values in non-fixed_at numerical or null 'at' clauses. It deletes whatever was there and substitutes something of the form 'at(0x000)'. It then, optionally, prints the memory map mentioned above.

amc is designed to take existing legal microprograms, using either entirely numerical addresses, or a mixture of numeric and symbolic addresses. Programs are more easily maintained, since one does not need to worry about moving instructions out of the way of additions, or worry about *filling in gaps*.

To write new programs or add new code to existing programs most easily follow these conventions:

- specify all fixed_at locations numerically.
- omit all other numbers; instead use 'at()', or if you have comments following the at clause, use 'at(   )' [5 spaces]; do not use 'at(0)'.
- indicate all dependencies clearly, for example 'at(abc+2)'.
- omit the 'at' clause entirely in simple, straight-line coding. This implies a dependency (this instruction must follow the previous instruction), which may not exist; however, this is normally not a problem in getting the program to fit in memory, and reduces the visual complexity of the microcode substantially. In most cases the instruction label and the branch in the previous instruction can also be omitted. Do not add a 'next' clause; the next clause is intended to warn that a branch would conflict with something in the microword.

- at microinterrupt entry points, use the new keyword assertion 'interrupt'. amc needs this to understand that this instruction does not fall through, even though it has the same format as instructions which normally do fall through.

If these conventions are followed, amc will do all the work of assigning the instruction addresses. In addition to saving the programmer a good deal of time, the checking incorporated in amc detects some bugs which the microassembler cannot catch, and the automatic assignment avoids the errors inherent in doing the assignments manually.

2.7 Summary

The microcode assembler/amc combination has proven to be a successful development tool. With it, we have developed microcode for five micromachines in the Butterfly Multiprocessor. By using a single microcode assembler we have reduced our software tools development cost significantly, and reduced maintenance cost as well. The microcode assembler has proven flexible and has helped to locate microcode logic errors before the microcode was installed. Because of its generality, we expect to be able to use the microcode assembler in future projects as well, both on the Butterfly Multiprocessor and on future machines.

DISTRIBUTION OF THIS REPORT

Defense Advanced Research Projects Agency
Dr. Robert E. Kahn (2)
Dr. Vinton Cerf (1)

Defense Supply Service -- Washington
Jane D. Hensley (1)

Defense Documentation Center (12)

USC/ISI
Dr. Danny Cohen (2)

MIT/Lincoln Labs
Clifford J. Weinstein (3)

SRI International
Earl Craighill (1)

Rome Air Development Center/RBES
Neil Marples (1)

Defense Communications Agency
Gino Coviello (1)

Bolt Beranek and Newman Inc.
Library
Library, Canoga Park Office
R. Bressler
R. Brooks
P. Carvey
P. Castleman
G. Falk
F. Heart
M. Hoffman
M. Kraley
W. Mann
J. Pershing
R. Rettberg
E. Starr
E. Wolf